CUTE and jCUTE : Concolic Unit Testing and Explicit Path Model-Checking Tools (Tools Paper)

Koushik Sen and Gul Agha
University of Illinois at Urbana-Champaign, USA.
{ksen,agha}@cs.uiuc.edu

Abstract. CUTE, a Concolic Unit Testing Engine for C and Java, is a tool to systematically and automatically test sequential C programs (including pointers) and concurrent Java programs. CUTE combines concrete and symbolic execution in a way that avoids redundant test cases as well as false warnings. The tool also introduces a race-flipping technique to efficiently test and model check concurrent programs with data inputs.

1 Introduction

Software testing is the primary technique used in the software industry to improve reliability, safety, security, and robustness of software. Our research on concolic testing [1, 6, 4] shows that we can combine random testing and symbolic testing of a program to provide a scalable tool for automatically generating test cases, which improves test coverage and avoids redundant test cases as well as false warnings. Concolic testing involves *explicit path model-checking* in which our goal is to generate data inputs and schedules that would exercise all feasible execution paths of a program. We have developed two automated concolic testing tools: CUTE for C and jCUTE for Java programs.

We have used CUTE and jCUTE to find bugs in several real-world software systems including SGLIB, a popular C data structure library used in a commercial tool, implementations of the Needham-Schroeder protocol and the TMN protocol, the scheduler of Honeywell's DEOS real-time operating system, and the Sun Microsystems' JDK 1.4 collection framework.

2 Concolic Testing

We briefly describe the algorithm for concolic testing; details can be found in [6, 5, 4]. The algorithm executes a program both concretely and symbolically. The symbolic execution differs from traditional symbolic execution, in that the algorithm follows the path that the concrete execution takes. During the execution, the algorithm collects the constraints over the symbolic values at each branch point (i.e., the *symbolic constraints*). At the end of the execution, the algorithm has computed a sequence of symbolic constraints corresponding to each branch point. We call the conjunction of these constraints a *path constraint*. Observe that all input values that satisfy a given path constraint will explore the same execution path, provided that we follow the same thread schedule.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
2006	**N/A**	**-**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
CUTE and jCUTE : Concolic Unit Testing and Explicit Path Model-Checking Tools (Tools Paper)	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Illinois at Urbana-Champaign Department of Computer Science 201 N. Goodwin Avenue Urbana IL 61801**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT
CUTE, a Concolic Unit Testing Engine for C and Java, is a tool to systematically and automatically test sequential C programs (in- cluding pointers) and concurrent Java programs. CUTE combines con- crete and symbolic execution in a way that avoids redundant test cases as well as false warnings. The tool also introduces a race-flipping technique to efficiently test and model check concurrent programs with data inputs.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**SAR**	**4**	

Apart from collecting symbolic constraints, the algorithm also computes the race condition (both data race and lock race) between various events in the execution of a program, where, informally, an event represents the execution of a statement in the program by a thread.

The algorithm first generates a random input and a schedule, which specifies the order of execution of threads. Then the algorithm does the following in a loop: it executes the code with the generated input and the schedule. At the same time the algorithm computes the race conditions between various events as well as the symbolic constraints. It backtracks and generates a new schedule or a new input, either by re-ordering the events involved in a race or by solving symbolic constraints, respectively, to explore all possible distinct execution paths using a depth first search strategy. Note that because the algorithm does concrete executions, it is sound, i.e. all bugs it finds are real.

There is one complication: for some symbolic constraints, our constraint solver may not be powerful enough to compute concrete values that satisfy the constraints. To address this difficulty, such symbolic constraints *are simplified by replacing some of the symbolic values with concrete values*. Because of this, our algorithm is complete only if given an oracle that can solve the constraints in a program, and the length and the number of paths is finite.

3 Tool Details

The tools, CUTE and jCUTE, consist of two main modules: an instrumentation module and a library to perform symbolic execution, to solve constraints, and to control thread schedules. The instrumentation module inserts code in the program under test so that the instrumented program calls the library at runtime for performing symbolic execution. jCUTE comes with a graphical user interface (a snapshot can be found in Figure 1).

CUTE and jCUTE uses CIL [3] and the SOOT compiler framework [8] to instrument C and Java programs, respectively. Instrumentation of jCUTE associates a semaphore with each thread and adds operations on these semaphores before each shared-memory access. These semaphores are used to control the schedule of the threads at runtime. To solve arithmetic inequalities, the constraint solver of CUTE uses lpsolve [2], a library for integer linear programming. CUTE and jCUTE save all the generated inputs and the schedules (in case of jCUTE) in the file-system. As such the users of CUTE and jCUTE can replay the program to reproduce the bugs. The replay can also be performed with the aid of a debugger. For sequential programs, jCUTE can generate JUnit test cases, which can be used by the user for regression testing as well as for debugging. jCUTE also allows the users to graphically visualize the multi-threaded execution.

CUTE provides a macro `CUTE_input(x)`, which allows the user to specify that the variable `x` (of any type, including a pointer) is an input to the program. This comes in handy to replace any external user input, e.g., `scanf(``%d'',&v)` by `CUTE_input(v)` (which also assigns a value to `&v`). Note that this macro can be used anywhere in the program. jCUTE also provides a similar function to obtain input from the external environment.

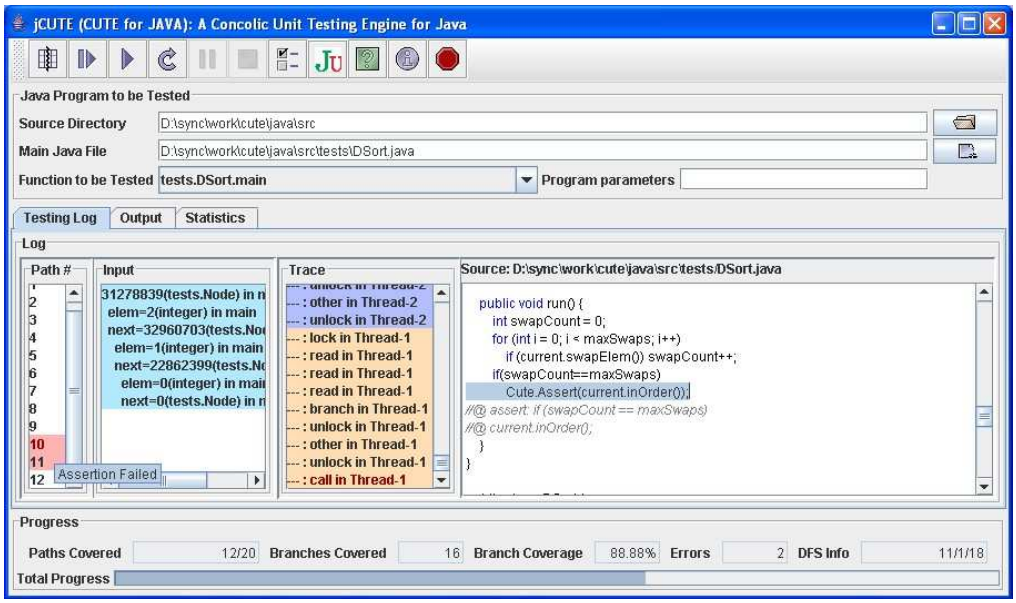

Fig. 1. Snapshot of jCUTE

4 Case Studies

We briefly describe our experience with two of the case-studies we have done, one is a data structure library in C and the other is the thread-safe Collection framework provided with Sun Microsystems' Java 1.4.

SGLIB Library. We applied CUTE to unit test *SGLIB* [7] version 1.0.1, a popular, open source C library for generic data structures, such as lists, hash tables, red-black trees, and so on. The library has been extensively used to implement the commercial tool Xrefactory.

We found *two bugs* in SGLIB using CUTE within 3 seconds of testing. The first bug is a segmentation fault that occurs in the doubly-linked-list library when a non-zero length list is concatenated with another zero-length list. The second bug is an infinite loop, which CUTE discovered in the hash table library. We reported these bugs to the SGLIB developers, who confirmed that these are indeed bugs. Further details about this case study along with branch coverage, runtime for testing, number of inputs generated, etc., can be found in [6].

Sun Microsystems' Java Collection Framework. We tested the thread-safe Collection framework implemented as part of the `java.util` package of the standard Java library provided by Sun Microsystems. A number of data structures provided by the package `java.util` are claimed as thread-safe in the Java API documentation. This implies that multiple invocation of methods on the objects of these data structures by multiple threads must be equivalent to a sequence of serial invocation of the same methods on the same objects by a single thread.

We chose this library as a case study primarily to evaluate the effectiveness of our jCUTE tool. As Sun Microsystems' Java is widely used, we did not

expect to find potential bugs. Much to our surprise, we found several previously undocumented data races, deadlocks, uncaught exceptions, and an infinite loop in the library. Note that, although the number of potential bugs is high, these bugs are all caused by a couple of problematic design patterns used in the implementation. The details of this case study can be found in [5]. Here we briefly describe an infinite loop that jCUTE discovered in the synchronized `LinkedList` class. We present a simple scenario under which the infinite loop happens. We first create two synchronized linked lists `l1` and `l2` by calling `Collections.synchronizedList(new LinkedList())` and add `null` to both of them. Then we concurrently allow a new thread to invoke `l1.clear()` and another new thread to invoke `l2.containsAll(l1)`. jCUTE discovered an interleaving of the two threads that resulted in an infinite loop. However, the program never goes into an infinite loop if the methods are invoked in any order by a single thread. jCUTE also provided a trace of the buggy execution. This helped us to detect the cause of the bug. A summary of the results of testing various Java synchronized Collection classes is provided in Table 1.

Acknowledgment This work is supported in part by the ONR Grant N00014-02-1-0715, the NSF Grant NSF CNS 05-09321, and the Motorola Grant RPF #23.

Name	Run time in seconds	# of Paths	# of Threads	% Branch Coverage	# of Funs Tested	# of Bugs R/D/L/E
Vector	5519	20000	5	76.38	16	1/9/0/2
ArrayList	6811	20000	5	75	16	3/9/0/3
LinkedList	4401	11523	5	82.05	15	3/3/1/1
LinkedHashSet	7303	20000	5	67.39	20	3/9/0/2
TreeSet	7333	20000	5	54.93	26	4/9/0/2
HashSet	7449	20000	5	69.56	20	19/9/0/2

Table 1. Results for testing synchronized Collection classes of JDK 1.4. R/D/L/E stands for data race/deadlock/infinite loop/uncaught exceptions

References

1. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, 2005.
2. lp_solve. `http://groups.yahoo.com/group/lp_solve/`.
3. G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and transformation of C Programs. In *Proceedings of Conference on compiler Construction*, pages 213–228, 2002.
4. K. Sen and G. Agha. Automated systematic testing of open distributed programs. In *Fundamental Approaches to Software Engineering (FASE'06)*, volume 3922 of *LNCS*, pages 339–356. Springer, 2006.
5. K. Sen and G. Agha. Concolic testing of multithreaded programs and its application to testing security protocols. Technical Report UIUCDCS-R-2006-2676, UIUC, 2006.
6. K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *5th meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2005.
7. SGLIB. `http://xref-tech.com/sglib/main.html`.
8. R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135.